Appendix D

High Speed Fiber Optic Interfaces

b y

Dr. Fouad Tobagi
Stanford University

Design and Implementation of Interface Units for High Speed Fiber Optics Local Area Networks and Broadband Integrated Services Digital Networks

İsmail Dalgıç Joseph Pang
Fouad A. Tobagi
Department of Electrical Engineering,
Stanford University, Stanford, CA94305-4055

31 October 1990

Abstract

This constitutes the final report for the portion of the JSC/Stanford Cooperative Research Program dealing with High-Speed Fiber Optic Networking. More explicitly, the research effort in this area comprised the design and implementation of interface units for high speed fiber optic local area networks (FOLANs) and Broadband Integrated Services Digital Networks (B-ISDN). Since we initiated this work several years ago, a number of network adapters that are designed to support high speed communications have emerged. We have been examining these proposals very closely, in order to identify their critical features, to build upon their experiences and to assess their appropriateness for very high speed communication in the Gb/s range. Our approach to the design of a high speed network interface unit has been to implement packet processing functions in hardware, using VLSI technology. We describe the VLSI hardware implementation of a buffer management unit (BMU), which is required in such architectures.

1 Introduction

An unfortunate problem with high speed networking today is the limited throughput that can be attained by the users in comparison with the available bandwidth of the transmission medium. This limitation exists even when only a single pair of users are actively communicating over the network, and stems from the complexity of the protocols used at the various layers (namely 2-4), and their implementation (typically in software). With the use of fiber optic technology and efficient multi access protocols, the transmission medium's bandwidth of local area networks has been increasing. Today, the FDDI standard is designed to run at 100 Mb/s. Furthermore, a great deal of research effort is under way to support broadband multimedia communications over a wide area network. An effort within CCITT is already underway to standardize Asynchronous Transfer Mode (ATM), which is the most appropriate switching technique to support communications over such a broadband integrated services network. At present, the ATM standard specifies line speeds equal to 155.520 Mb/s, 620.080 Mb/s, and above. Consequently, numerous applications have emerged with very high bandwidth requirements, such as the transmission of high resolution images or video signals, communications among remote supercomputers to perform jointly a task such as flight simulation, etc. Such applications require packet processing within the network interfaces at rates several orders of magnitude higher than that possible with current software implementation of network protocols. Accordingly, the problem of designing high speed network interface units has become a crucial one.

The throughput limitation in today's network interfaces is mainly due to the software implementation of the protocols. Indeed with the exception of the CRC and addressing functions at the logical link control (LLC) sublayer, all functions residing at the LLC sublayer and above are implemented in software, either in the host processor or using a microprocessor in a network adapter board. Furthermore, packets and their headers are stored in a shared memory which gets accessed several times while processing the same packet, thus complicating the memory management problem as well as introducing memory access delays.

In one of our earlier research efforts, we measured the performance of a program implementing the IEEE Std 802.2 Logical Link Control (LLC)

protocol in order to improve our understanding of packet processing requirement within a network interface [1]. Our results showed that most of the program execution time is spent in a few relatively low-level functions; most importantly, the block movement of data in memory and packet queue management. The importance of these findings was paramount as they suggested the main areas to be addressed in the design of high speed network interfaces. We are taking advantage of these findings in our research effort described here.

Since we initiated this work several years ago, a number of network adapters that are designed to support high speed communications have emerged. All of these network adapters were designed around an on-board microprocessor with the exception of the XTP Protocol Engine which is designed for VLSI implementation [2]. We have been examining these proposals very closely, in order to identify their critical features, to build upon their experiences and to assess their appropriateness for very high speed communication in the Gb/s range. This task is not yet complete, and we will pursue it in the future. At the present time, we are in a position to describe accurately the architecture underlying each adapter and the expected performance given the specification of their components. In Section 2 we give a brief description of three key architectures, namely; NAB [3], CAB [4] and XTP Protocol Engine. Not unlike XTP, our approach to the design of HSNIU has been to implement packet processing functions in hardware, using VLSI technology. One principal reason for this choice is that with VLSI hardware implementation of protocol functions, high rate on-the-fly processing may be achieved, thus overcoming many of the bottlenecks experienced in architectures based on software implementation. Another reason is that, with VLSI hardware implementation, one may incorporate multiple packet processors in the same user device, either to achieve higher throughput or to accomodate multiple distinct streams of data as will be required in multimedia applications, while keeping the interface design relatively compact and at low cost. In Section 3, we describe the steps accomplished so far in this endevour particularly the VLSI hardware implementation of a buffer management unit (BMU), required in such architectures.

2 Related Work

2.1 Introduction

In this section, we describe several existing high speed network adapters, explaining their operation and identifying the similarities and differences among them. We have chosen to study 3 such designs, namely, the VMP Network Adapter Board (NAB) [3], the Nectar CAB [4] and the XTP Protocol Engine [2]. Among the three, the NAB and the CAB are designed around a microprocessor-based architecture, whereas the Protocol Engine was designed with the VLSI implementation in mind.

In addition, we are still examining several other designs such as a network adapter being designed at the IBM Zurich Research Laboratory which is based on a multiple-processor architecture using transputers [5]. However, our studies of these designs are not finalized yet, and thus we are not discussing these designs here.

2.2 The VMP Network Adapter Board (NAB)

The NAB is a network adapter designed to support efficient distributed processing over a high speed network. It is designed for the VMP multiprocessing system, but it is applicable to other computer systems as well. It is designed to implement the VMTP transport protocol. However, without any major changes in its architecture, it can implement other transport protocols as well.

The internal architecture of the NAB consists of five major components, as shown in Figure 1. The following is a description of the functions of each component:

Network Access Controller (NAC): It implements the network access protocol, performs the data conversion between serial and 32-bit parallel forms, and transfers the data between the network and the packet pipeline.

Packet Pipeline: It consists of two separate pipelines, one for sending packets and one for receiving packets, thereby achieving full duplex communications. The sending packet pipeline generates transport level

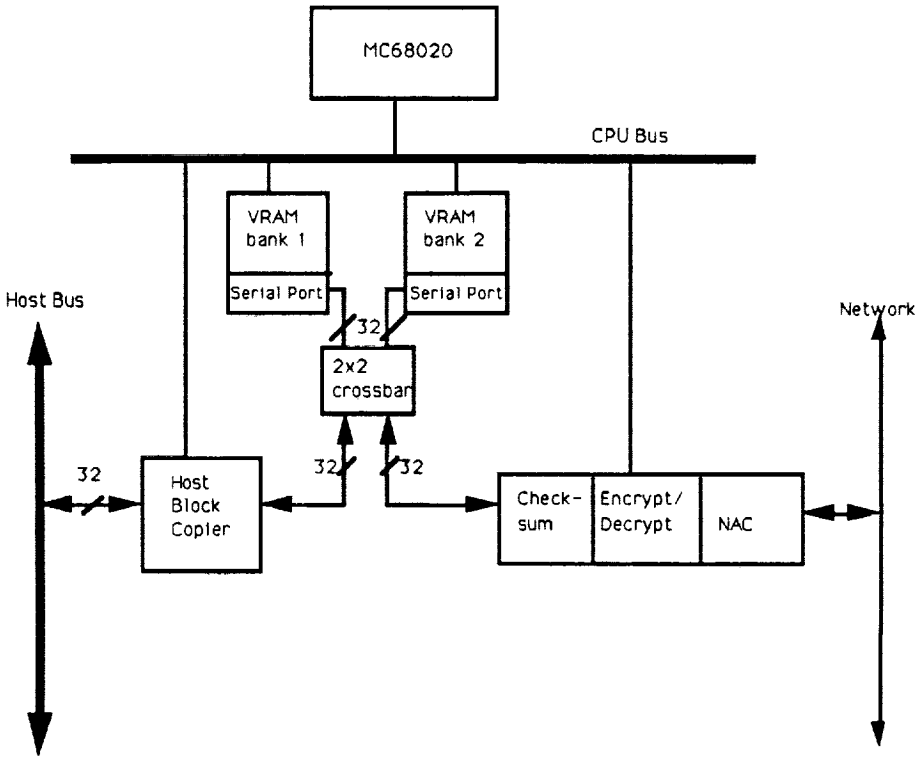


Figure 1: The NAB

checksums and optionally encrypts the data. Similarly, the receiving packet pipeline performs checksum calculations and if needed, decyrpts the incoming data. The packet pipelines operate at the network data rate and perform processing on the fly. In order to be able to do that, the checksum portion is placed at the end of a packet in VMTP. Checksumming and encryption are the only two functions that require accessing to the data portion of the packet, so using these pipelines, the memory access requirements for the on-board processor is greatly reduced as it would only need to access the header portion of a packet.

Buffer Memory: Video RAM ICs are used as buffer memory in the NAB. Such an IC provides two independently accessed ports: one provides the traditional random-access transfer, and the other provides high speed serial access through a 4-bit wide shift register. The organization of the VRAM chips used in the NAB design is shown in the Figure 2. 8 chips are used per memory bank to provide a word size of 32 bits in both the random access and serial ports, as also shown in that figure.

The serial access port does not need address setup and decode time; so it is faster than the random access port. In the NAB prototype, the serial access time is 40ns per word and the random access time is 200 ns, giving an effective transfer rate of 800 Mb/s over the serial port and 160 Mb/s over the random access port. Because of this difference in the bandwidth, the serial access port is used for both transferring the data between the network and the buffer memory and between the host computer and the buffer memory. The random access port is only used by the on-board processor for protocol processing. The transfers between both the buffer memory and the host and the buffer memory and the network have a bursty nature since an entire packet is transferred at a time. Therefore, the choice of using the serial port for these transfers is appropriate.

In order to be able to receive a packet coming from the packet pipeline into the buffer memory at the same time as writing a previously received packet from the buffer memory into the host (or vice versa), the buffer memory is organized as two separate banks. A 2x2 crossbar switch connects both of these banks to both the packet pipeline and the host adapter so that a packet can be copied from the packet pipeline into

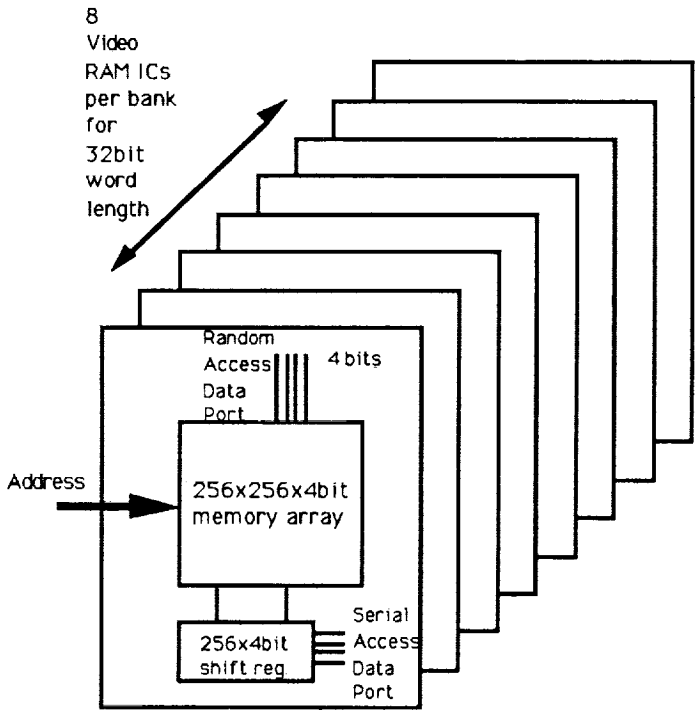


Figure 2: VRAM buffer memory organization for one memory bank

one of the memory banks, and then by changing the state of the switch, it can be copied to the host, while another packet is being received into the other buffer memory bank. This effectively doubles the bandwidth of the serial port. The state of the crossbar is controlled by the on-board processor.

On-board processor: The purpose of the on-board procesor is to manage the buffer queue and implement the transport level protocol, and also to control the operation of the host block copier and the packet processing pipeline. A 16-Mhz MC68020 was chosen for this purpose, which has a rating of 2 MIPS.

Host Block Copier: Its purpose is to transfer data between the host computer and the NAB. It includes a DMA controller to efficiently copy data from the host memory into the NAB buffer memory. There is a 1024 byte control register in this unit which accessible by both the NAB and the host computer. This register is used to exchange requests and responses between the NAB and the host. When sending packets, two different schemes are used depending on the size of the packet. For long packets that would not fit into the control register, the host places the pointers into the control registers for the DMA controller to get the data from the memory. For short packets, the host writes the data into the control register directly without invoking the DMA controller on the NAB. This scheme is motivated from the idea that a short packet to be transmitted is most likely to be a message that is very recently prepared, and hence a copy of it is in the host processor cache. Thus, instead of putting it back into the main memory and then using the DMA controller, it is more efficient to copy the data directly from the cache into the control register.

When receiving a packet from the network, the NAC performs the serial to parallel conversion and starts transfering the received packet into the decryption hardware in the packet pipeline. Also, it interrupts the on-board processor to notify it about the packet arrival. At the first decryption stage, a decryption key is searched for, and if it is located, the data proceeds through the decryption stages. Otherwise, the decryption stages are disabled and the data proceeds directly to the checksumming stage. If the packet passes the

checksumming correctly, it is transferred to the shift register in one of the memory banks, depending on the state of the crossbar switch. In exceptional cases, such as the buffer memory being full or the packet not having a correct checksum, the packet is lost and the packet pipeline is flushed and restarted. This is handled by the on-board processor.

When the entire packet is moved into the shift register, it is transferred into an available location in the random access memory specified by the on-board processor, and queued there for header processing. After the header processing is completed, the packet is trasnferred to the host memory using the host block copier.

For sending a packet, the host writes the request to the control register (along with the data for a short packet, and the pointers to the data for a long packet). As soon as the control register is written, its contents are transferred to a queue of requests for the attention of the on-board processor. While the data of the packet is being transferred into the buffer memory, the on-board processor prepares its header. When the header processing is finished, the packet is transferred to the packet pipeline, by putting the crossbar in the opposite state as it was when the packet was moved into the memory and then copying the packet to the pipeline through the shift register. When the entire packet is passed from the pipeline and transmitted over the network, the NAC interrupts the on-board processor which in turn returns the packet buffer to the pool of empty buffers. If a packet is being received from the network while another packet to be sent is being transferred to the buffer memory, the received packet is queued at a stage provided between the packet pipeline for reception and the shift register of the buffer memory. Other than this situation, a packet that is to be sent is delayed until there is no packet being received, i.e., the priority is given to the received packets.

It is clear that for the header processing function not to be the bottleneck, the on-board processor must finish processing one packet header in a time shorter than the time to receive an entire packet. The maximum network bandwidth at which this can be achieved depends on the relative bandwidths of the serial and random access ports, the on-board processor speed and the length of the packets, as well as the header processing requirements of the particular protocol being implemented. Another bandwidth limitation comes from the packet pipeline. In the current prototype, the packet pipeline is able to operate at data rates around 100 Mb/s.

2.3 The Nectar Communication Accelerator Board (CAB)

Nectar is an experimental network for multiprocessing, built to support the WARP operating system. It consists of a number of crossbar switches, called HUBs, and network adapter boards, called CABs.

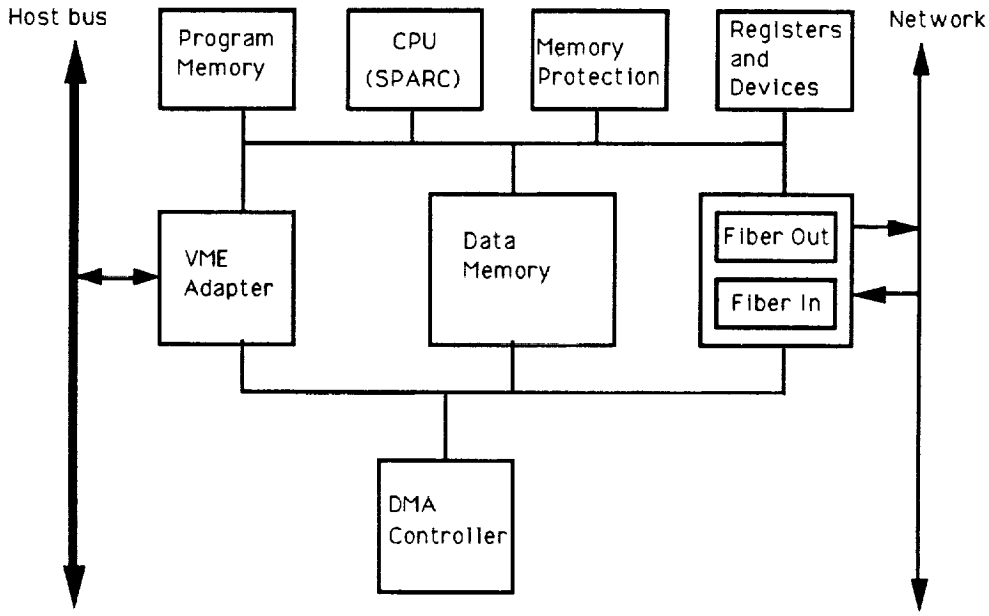


Figure 3: The CAB

Just like the NAB, the CAB is a processor-based design. It is designed to be general enough to implement several transport protocols. It consists of 8 main parts which are interconnected as shown in Figure 3. The following is a description of these parts:

CPU: It is used to implement most of the protocol processing, and also it controls the operation of the CAB. In the CAB prototype, a 16Mhz SPARC processor is used for this purpose.

Registers and Devices: This part consists of various devices to support high speed communications such as a hardware checksum device and hardware timers. Also, some of the registers to store the state of the

CAB are included in this part. Differently from the NAB, these devices are placed on the CPU bus, providing flexibility at the expense of additional data memory bandwidth due to the carrying of the data from the memory to these devices over the CPU bus.

Program Memory: This is the memory that is intended to be used only for storing the programs for the CPU. A fast static RAM (35ns) is used for this purpose so that the peak execution rate of the processor can be sustained without any caching.

Data Memory: It is used as the buffer memory for both the received and sent packets. In the prototype implementation, a 35ns dual ported static RAM is used as the buffer memory, with one port connected to the data memory bus and the other to the CPU bus. The CPU bus is used only for the protocol processing by the CPU which includes copying the entire data portion of the packet over this bus to the hardware checksum device. The queues in this memory are managed by the CPU.

Fiber Out/Fiber In Queues: They consist of the media access units for the sent and received packets, respectively. TAXI chips are used to perform the serial to parallel conversion. Also, FIFO queues are provided here for both incoming and outgoing packets.

DMA Controller: A DMA controller resides on the data memory bus and it is normally used to transfer the packets between the network and the data memory as well as between the memory and the host. The DMA controller also waits for data to arrive if the input queue is empty, or for data to drain if the output queue is full, thereby performing some form of flow control.

VME Interface: It provides an interface for both the data memory bus and the CPU bus to the host bus, which is assumed to be a VME bus. Through this interface, both the program and the data memories on the CAB are directly addressable by the host, and so are all registers and devices of the CAB. In addition, the CAB CPU can access the host resources through this interface.

Memory Protection: The CAB is intended to offload the host of application tasks when it is not performing communication functions. In order to support this it runs a simple multitasking kernel which requires a memory protection mechanism that is provided in the CAB. Currently it supports up to 32 protection domains.

The communication between the host and the CAB is normally accomplished using DMA, shared buffers and VME interrupts. However, it is also possible for the host to bypass the CAB by performing all of the protocol processing by itself and accessing the **Fiber In** and **Fiber Out** queues directly. In the following, we will describe the former mode of operation.

The DMA controller continually polls the **Fiber In** queue for new packets. When a new packet is received, after the serial to parallel conversion of the packet is performed and the packet is buffered in the **Fiber In** queue, it is transferred by the DMA controller to a memory location in the data memory that is provided by the CPU, where it is queued for protocol processing. The CPU performs the processing of the packet header, which is dependent on the particular protocol being executed. Meanwhile, the data portion of the packet is transferred over the CPU bus to the hardware checksum device, either by the CPU itself or by the checksum device which incorporates a simple DMA mechanism.

When the processing of the packet is finished, the data is copied to the host memory using the DMA controller. Therefore, the packet gets moved twice over the data memory bus and once over the CPU bus. Additionally, on the CPU bus, the packet header is accessed for protocol processing. This access volume is likely to be less than the access volume for transferring the packet once over the bus for reasonably long packets. As a result, an upper bound on the throughput that can be achieved is half of the bandwidth of the data memory bus.

The sending of the packets proceeds in an analogous fashion. The CPU is notified of a send request by the host by placing a request message in a special mailbox in the CAB data memory. Then the CPU sets up the DMA controller to transfer the data from the host memory to the data memory. After the packet processing is completed, which now includes packetization and hardware checksum generation, the DMA controller is set up by the CPU to transfer the packet from the data memory to the **Fiber Out** queue.

Overall, the CAB architecture does not seem to be too much different from the NAB architecture. The main differences are the following: (i) in the CAB, the checksum device is placed on the CPU bus, resulting in extra memory accesses on the bus, whereas in the NAB it is placed between the network access controller and the buffer memory; (ii) the CAB memory and devices are accessible to the host, thereby allowing the host to produce and consume data directly in the CAB data memory; (iii) support is provided to offload the host applications to the CAB CPU when it is not performing networking functions.

2.4 The Protocol Engine

Contrary to the other network adapters, the protocol engine is designed to be implemented as a VLSI chip set. It executes a simplified transport protocol called XTP, which is optimized for receiver performance, as normally the receiver part of a protocol constitutes a bottleneck. It is intended to perform the packet processing on-the-fly, in real time.

The architecture of the protocol engine consists of seven main parts as shown in Figure 4. The following is a description of each part:

Protocol Engine Control Logic: Along with the address logic and buffer logic, the protocol engine control logic forms the core of the system. It contains a state machine for handling the transport protocol and controllers for the address, network, buffer and host components. The XTP is connection oriented and each connection has an associated state vector stored in the protocol engine. The protocol engine control logic is responsible for managing these state vectors, performing sequence number checks, packetization, observing round-trip delays, adjusting connection timers accordingly and performing retransmissions.

Address Logic: Instead of sending a long address associated with a connection with every packet, it is only sent in the header of the first packet along with a key. In the subsequent packets, only the key is sent in the header, implying the long address. When a packet is received by the protocol engine, the address logic searches for the long address in the route table and state vector memory, and if it is found, loads the state vector corresponding to that connection to the control logic. The

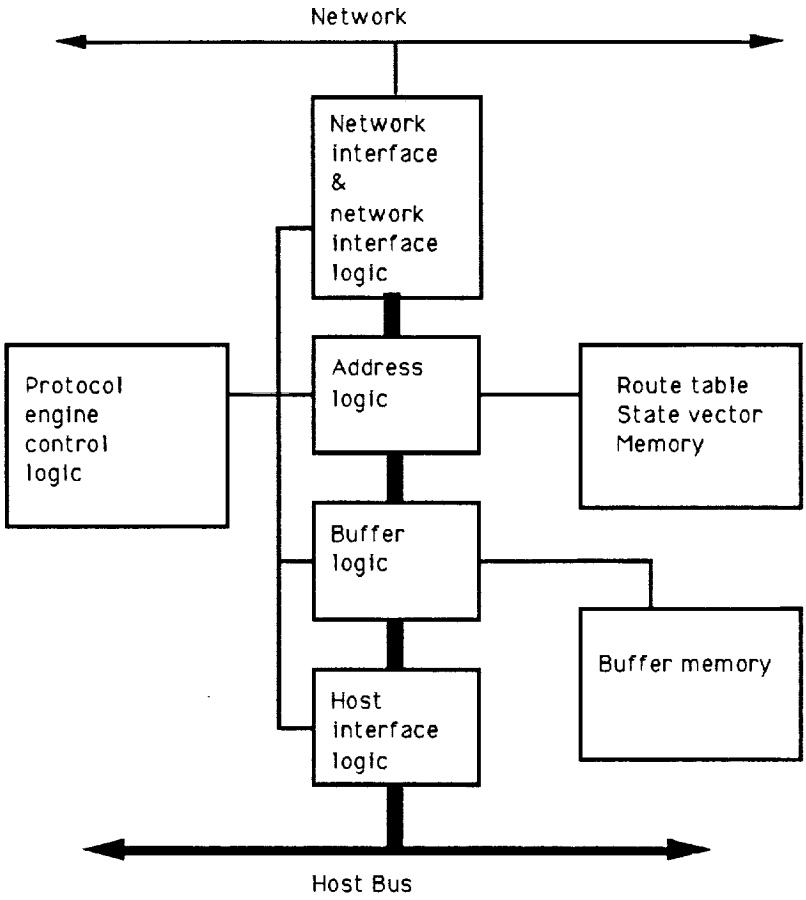


Figure 4: The Protocol Engine

address logic also is used to store the updated state vectors into the route table and state vector memory.

Route Table and State Vector Memory: This memory is used to store the route table, long addresses and the state vectors corresponding to the connections. It is managed by the address logic as mentioned above.

Buffer Memory: It is the place where the incoming and outgoing packets are queued. Similarly with the NAB, VRAM chips are used for this purpose where the serial port is used for the network and host accesses and the random access port is used for protocol processing.

Buffer Logic: It is used for transferring the data between the buffer memory and the protocol engine control logic, network interface logic and host interface logic. It is also used for managing the multiple queues in the buffer memory.

Host Interface Logic: This component is used for moving the data between the protocol engine and the host computer. Its design is dependent on the particular host bus on which the protocol engine is being used. However in most cases, it would contain a DMA controller to copy the data between the host memory and the protocol engine buffer memory.

Network Interface Logic: This part performs the data transfers between the network and the protocol engine, and it implements the media access protocol. It also performs the checksumming function and serial to parallel conversion. The frame check sequence is stored at the end of the packet in XTP, hence allowing the checksumming to be done on the fly.

When receiving a packet, the network interface logic begins to transfer the packet to the address logic as it comes in. As soon as the header gets into the address logic, the address logic uses the address key in the header to find the corresponding state vector for the packet's connection. While this is being done, the incoming data field of the packet gets to be stored into the buffer memory. Also, the computation of the frame check sequence (FCS) gets performed as the data passes through the network interface logic. If the

state vector is located, it is loaded into the control logic. Otherwise, this is either a new connection, in which case a new state vector is generated, or it is a faulty packet and it is rejected. After the state vector is loaded, the control logic checks the sequence numbers against the expected sequence numbers. At the time when this is finished, the packet is completely received and buffered, and the FCS is received. If the FCS checks out correctly and the sequence numbers conform with the expected ones, then the packet is linked to the appropriate queue and the state of the corresponding connection is updated and stored. After this, the host is notified about the arrived packet and the data is copied into the host memory by using the host interface logic.

When sending a packet, it is first copied through the host interface to the buffer memory and linked in the queue that corresponds to the belonging connection. Then the state vector for that connection is loaded into the control logic which in turn prepares the packet header. Then it updates the state vector and sets up the timer corresponding to this connection. After this, the packets begins to get transmitted on the network through the network interface logic. As the data passes by, the network interface logic computes the FCS and appends it to the end of the packet.

3 The High Speed Network Interface Unit (HSNIU)

3.1 Architecture

3.1.1 Introduction

Our goal is to design and implement a high speed network interface unit (HS-NIU) capable of providing network users with the throughput required for broadband applications. Furthermore, we are seeking a design which can be useful for a wide range of applications with different end-to-end communications requirements. This effort requires the performance of three basic steps:

- The determination of all the functions that the interface must perform, with a clear identification of the critical ones;

- The determination of an architecture, according to which the functions are to be partitioned, which provides the desired modularity and achieves the required performance;
- The choice of appropriate technologies by which to implement the functions.

It should be clear that the three steps identified so far are closely interrelated, and that the process of devising an adequate solution to the design of an HS-NIU is an iterative one, comprising in-depth evaluations of many trade-offs and alternatives which cut across all three areas.

Preliminary investigation has shown that there are 5 basic critical factors which must be taken into account in the design. These are:

1. The medium's bandwidth
2. The host's bandwitdh
3. The interface's memory bandwidth
4. The packet processor's bandwidth
5. The medium access controller's reaction time.

Functions within the interface will undoubtedly be performed at different speeds, and it is the goal of an optimum design to keep in balance all the various elements performing these functions. Given the five factors and the differences among them, it appears logical to divide the interface into three basic units (see also Figure 5):

- The Medium Dedicated Unit (MDU)
- The Storage and Processing Unit (SPU)
- The Host Dedicated Unit (HDU)

The MDU will certainly have to run at the speed of the medium as far as data transfer is concerned, and will have to include a media access controller whose reaction time is compatible with the performance requirements. The

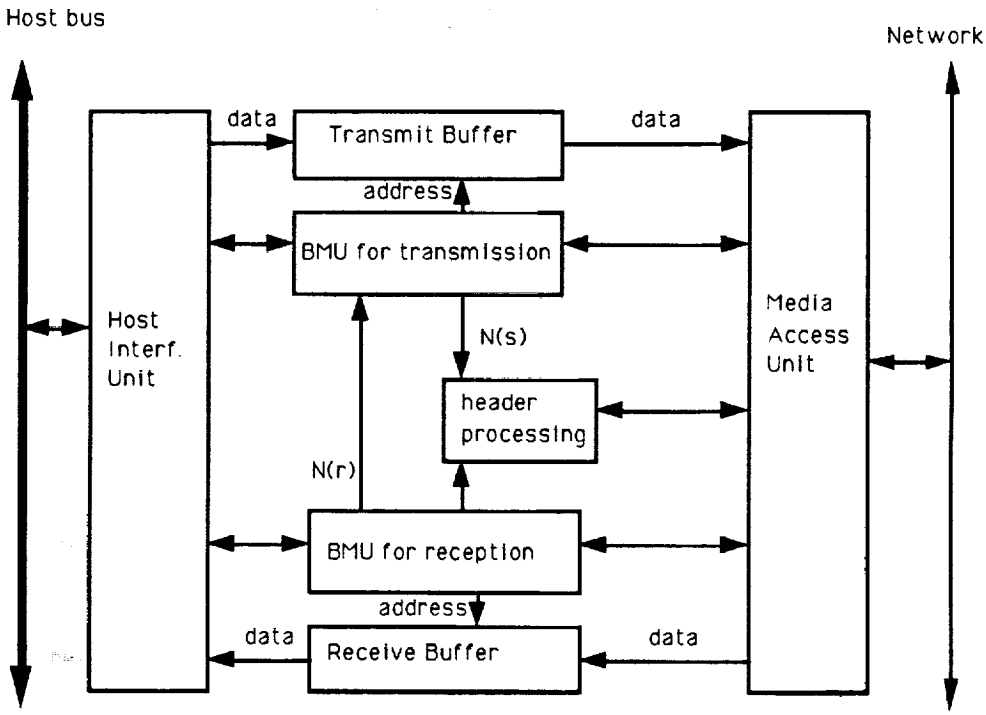


Figure 5: Block diagram of the HSNIU

MDU will perform parallel-serial conversions, and all functions that may be implemented serially, such as CRC and encoding. The MDU will be entirely implemented in hardware.

The SPU will provide the storage capability for user data, and will perform packet processing such as packetization, LLC and the transport protocol. We pay attention to the design of the SPU so that the memory bandwidth is sufficiently high to interface to the MDU and the HDU. Implementation of the processing unit may be a combination of hardware and firmware, depending on the specific functions implemented.

Finally, the HDU will provide the interface with the user host and the SPU. It will also be entirely implemented in hardware.

Of the three units mentioned above, it is considered that the SPU is the most crucial one and hence we concentrated on it as the main part of our research at this stage. Indeed it is the core of any interface regardless of the type of medium, access control protocol or host under consideration. Furthermore, it is the least experimented with, and it is the most challenging in determining the best architecture for a high speed interface.

As part of our effort on designing the SPU, we have designed the Buffer Management Unit for transmission at the logic level. We are currently working on designing the buffer management unit for reception. After this, the next step will be to implement a prototype of the SPU in VLSI.

The folowing parts of this section mainly cover the description of the Buffer Management Unit for transmission.

3.1.2 Functions of the Buffer Management Unit For Transmission (BMU-T)

The BMU-T serves to replace the most frequently executed routines in conventional software buffer management. After some careful considerations, it has been suggested that the BMU-T must perform the following 3 major functions:

- To manage the transmit dual-port buffer as a FIFO queue.
- To perform packetization on the fly.
- To handle data link Go-Back-N ARQ protocol.

3.1.3 Specific algorithms used in the Buffer Management Unit

Although there are interactions between the three functions of the BMU, they can be more or less discussed separately.

a) FIFO Management:

As mentioned before, the dual-port buffer is managed as a FIFO queue to avoid the addressing computations that are done in software management.

Depending on the application, the host will generate messages of variable lengths and forward them to the transmit buffer. Upon request from the host, the BMU-T must supply the appropriate addresses so that each message will be stored in a sequential manner.

Each message will be packetized while it is being stored in the buffer. The addresses defining the packet boundaries (the end address of each packet) will be stored and maintained in an internal FIFO queue and inside the BMU-T.

Packets will be removed from the buffer by the Media Access Unit sequentially. Again, the BMU-T will be responsible of supplying the addresses.

To implement the above algorithm, two address counters are required, one for the incoming data (from the host) and the other for the outgoing data (to the network). Furthermore, a number of supervisory signals are required. Figure 6 shows a block diagram of the FIFO management section.

The supervisory signals are *Full*, *Empty* and *EOP* (end of packet). *Full* indicates that the address is full and prevents the host from writing new data to the buffer. *Empty* indicates that the queue is empty[1] and prevents the Media Access Unit from reading from the buffer. *EOP* indicates that the current outgoing data word is the end of a packet.

To generate the supervisory signals *Full* and *Empty*, we need a number of status pointers for the address queue. They are TOQ, BOQ+1 and LACK. TOQ is the top-of-the-queue pointer, BOQ+1 points to the next position after the bottom-of-the-queue and LACK points to the last acknowledged packet. When BOQ+1 coincides with LACK, *Full* becomes true and when TOQ coincides with BOQ+1, *Empty* becomes true.

EOP is generated by comparing the outgoing address counter with the end address of the current packet.

[1] We shall see later that *Empty* may have another meaning.

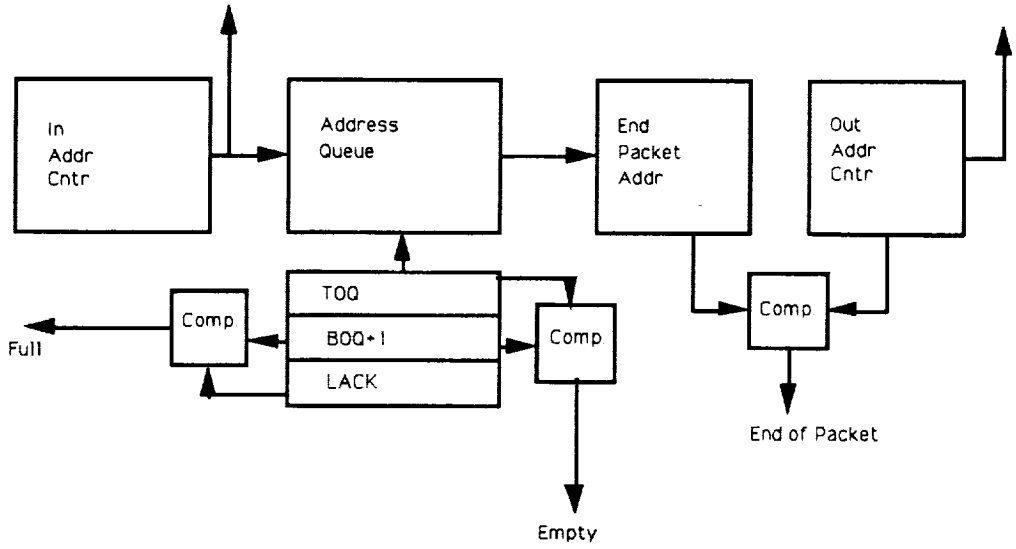


Figure 6: FIFO Management Section of the BMU-T

b) Packetization:

Messages will be broken up into as many maximum size packets as possible until the end of message is asserted by the host. Thus a message will contain a number of maximum length packets with possibly a short packet at the end.

c) Go-Back-N ARQ protocol:

To achieve reliable communication, the BMU-T will be required to execute the Go-Back-N ARQ protocol. In this project, we use a modified Go-Back-N ARQ protocol whereby each outgoing packet will be attached a sequence number N(S) and these packets will be acknowledged by a sequence number N(R). A new N(R) acknowledges all packets from the last N(R) to this new N(R) minus 1. Thus, the new N(R) is the expected sequence number at the other end of the communication path. Only a certain maximum number of packets can be outstanding without acknowledgement at any given time. When this is maximum is reached, a local timer is triggered. If the timer expires before an acknowledgement arrives, all the outgoing packets must be retransmitted. If an acknowledgement arrives, the timer will be reset and the normal operation resumes.

In view of the above protocol, the BMU-T must maintain outstanding packets by the pointer LACK. Moreover, if the number of outstanding packets is equal to the maximum possible, the BMU-T must stop the Media Access Unit by the signal *Empty*. This is why *Empty* may take on a different meaning from an empty queue. When the timer expires, a number of actions must be taken:

1. Reload TOQ by LACK+1.
2. Reload outgoing address counter.
3. Reload next end packet address.
4. Reload N(S).
5. Reset timer.
6. Disable any incoming N(R).

The details of implementing the timeout mechanism is rather complicated because so many actions have to be taken and not all of them can be done in the same cycle. We present the details in subsection 2.2 below.

3.1.4 An Integrated Block Diagram of the BMU-T

The result of the above considerations led us to the logic block diagram of the BMU-T as shown in Figure 7.

3.1.5 Testing Strategy

We shall use the LSSD testing technique. The scan path will include:

1. All counters
2. All pointers
3. Stored N(S) and N(R) values
4. End of packet address

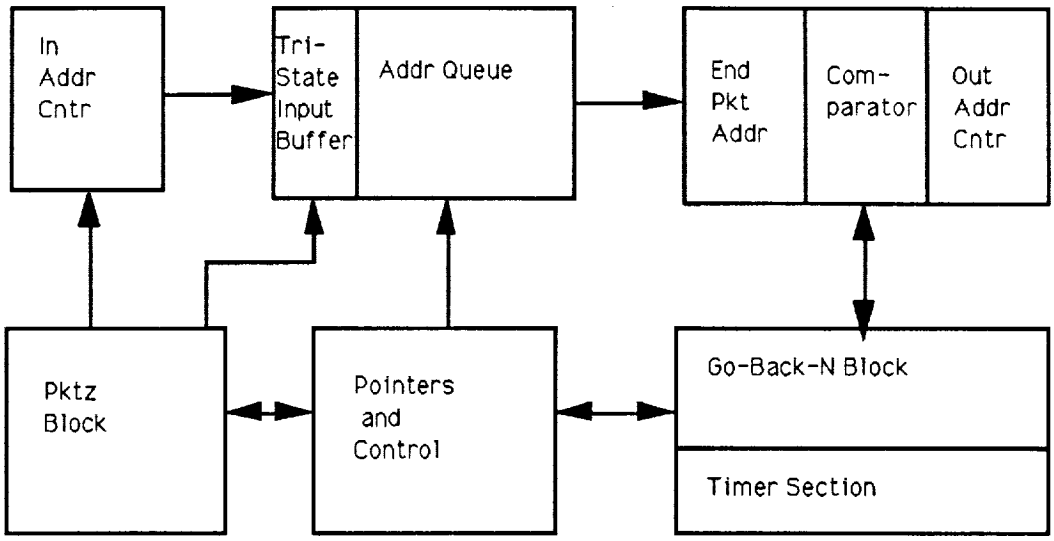


Figure 7: The logic block diagram of the BMU-T

By shifting in the appropriate test vectors, we can test:

1. All counters
2. All pointers
3. Logic generating external signals *Full*, *Empty* and *EOP*
4. Address queue cells (By writing through *in-addr-cntr* and reading from *next-end-pkt-addr*)
5. Packetization Logic
6. Go-Back-N Logic

3.2 Micro-architecture

3.2.1 Individual Block Descriptions

Figure 7 is an appropriate partition of the BMU-T. We describe these blocks in detail in this section.

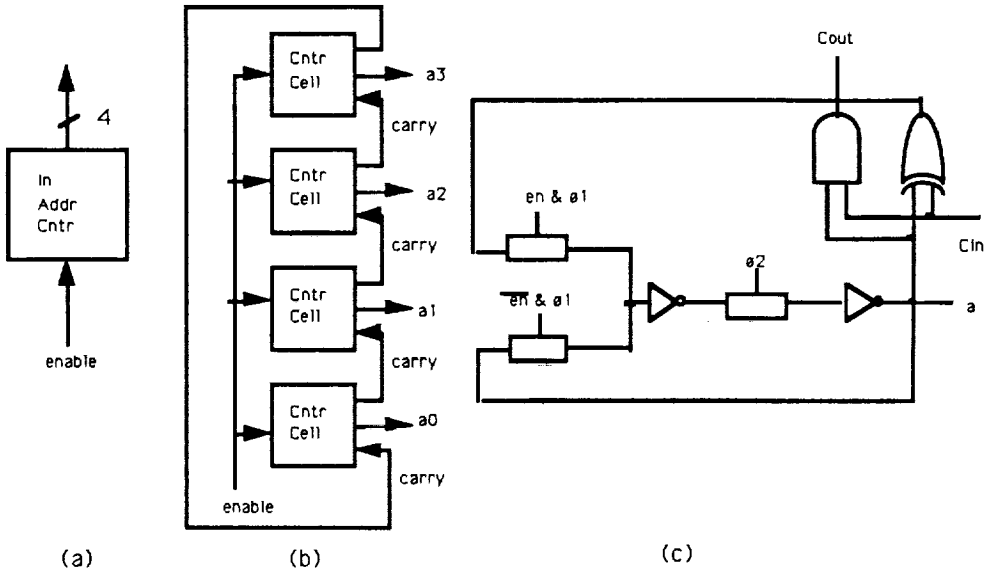


Figure 8: In Address Counter

In Address Counter: The *in-addr-cntr* is a 4-bit binary counter with an enable signal shown in Figure 8a. Figure 8b shows the wiring among the four counter cells and Figure 8c shows the schematic of each counter cell. When enable is high, the counter will advance but when enable is low, the counter outputs will remain the same.

The enable signal is generated by the packetization block by $host\text{-}req \cdot \overline{Full}$.

Address Queue with Tri-State Input Buffers: Figure 9a shows the Address Queue and the input buffer along with their sizes. The Address Queue has an input bus and two output busses. The input bus is a static bus while the outpus buses are precharged as shown in Figure 9b.

The write and read select signals come from the TOQ, BOQ+1, LACK pointers. The enable signal comes from the packetization block by *max-pkt* OR *EOM* (maximum length packet written or end of message).

Output Address Section: Figure 10a shows the sub-blocks and their sizes in the output address section. Figure 10b,10c and 10d show the schematics of the *next-end-pkt-addr*, comparator and the *out-addr-cntr* respectively.

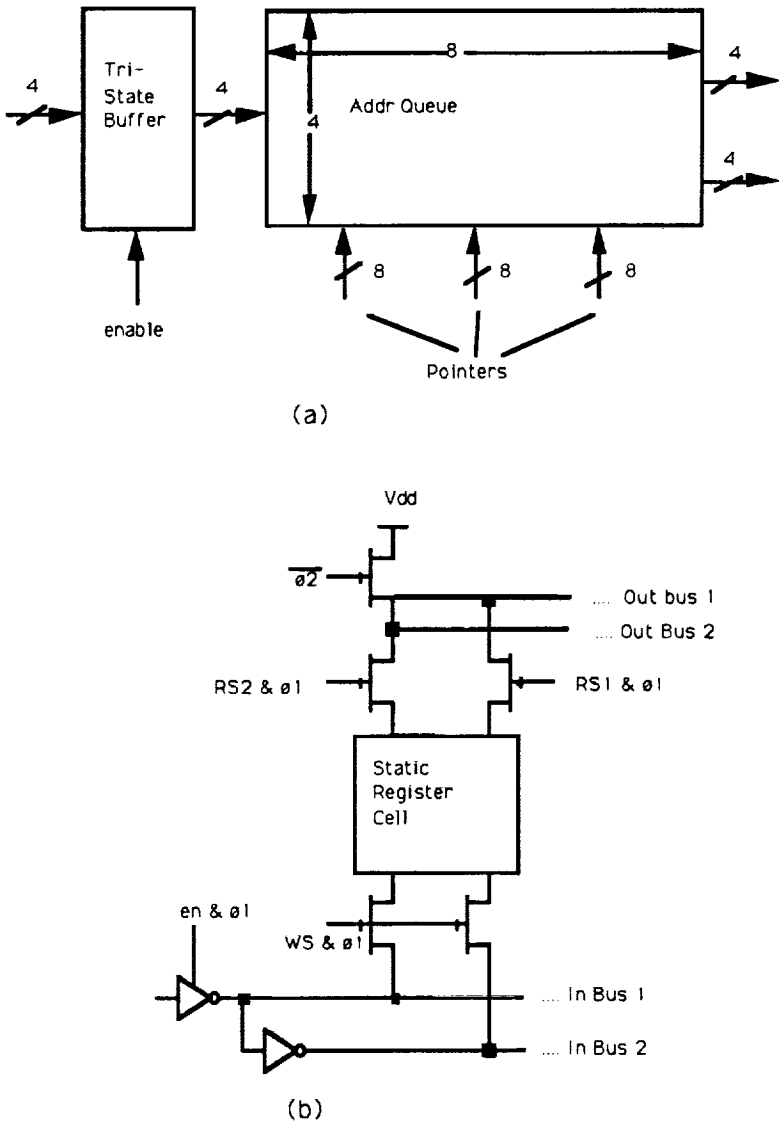


Figure 9: Address queue and the tri-state input buffer

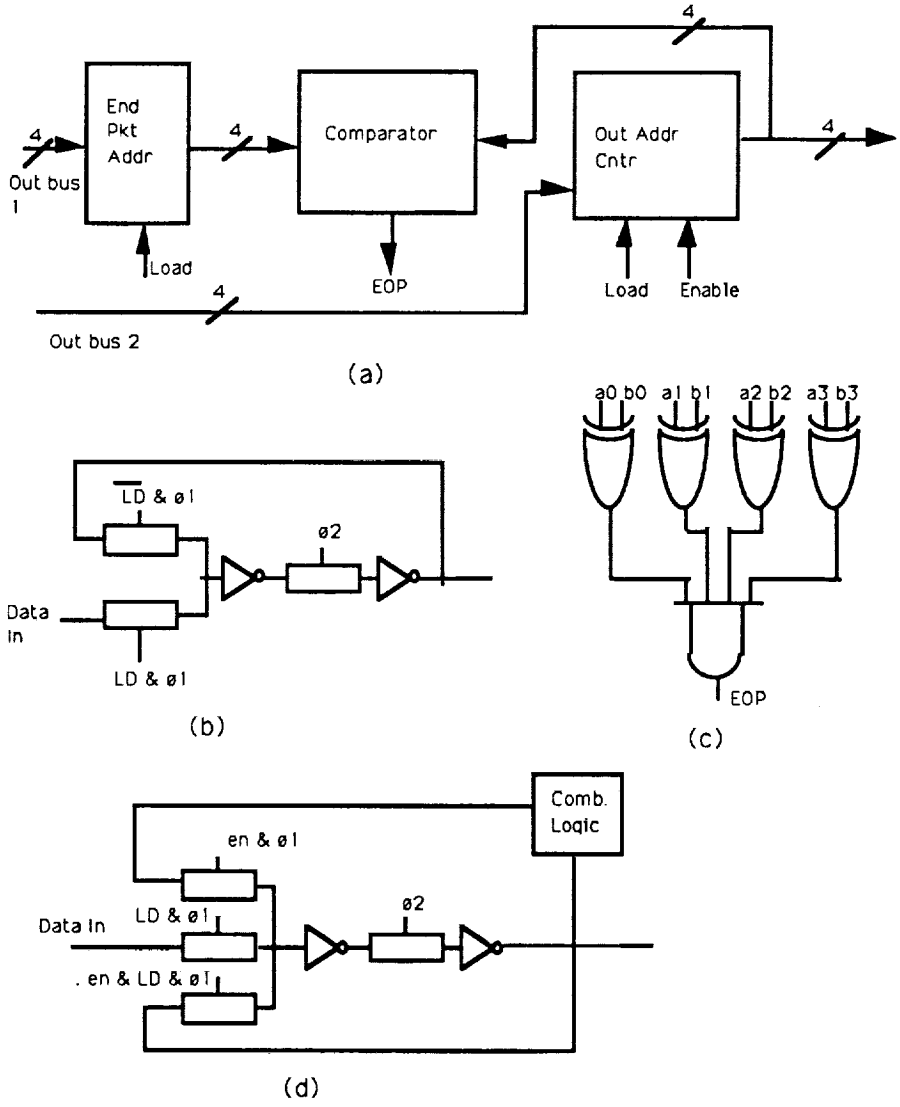


Figure 10: Output Address section

The load signal to *end-pkt-addr* is derived from *EOP*, *Empty* and timeout signal from the Go-Back-N block. The load signal to the *out-addr-cntr* is derived from the timeout signals. The enable signal is derived from *man-req*·$\overline{Empty}$ and also the timeout signals.

Packetization Block: Figure 11a shows the internal structure of the packetization block and Figure 11b shows the schematics of the *pkt-siz-cntr* cell. This cell is similar to the *in-addr-cntr* except there is a reset signal here. The *pkt-siz-cntr* is a 3 bit counter and the count of the most significant bit is the *cntr-full* signal. The *host-req* and the *EOM* are external signals from the host. The *Full* signal is generated from the pointer section.

The two output signals are used to enable the *in-addr-cntr*, the tri-state buffer and to advance the BOQ+1 pointer.

Pointers and Control: All pointers are 8-bit long (no encoding). Figure 12a show the internal structure of the pointer section. Figures 12b, 12c and 12d are the schematics for the cells in TOQ, BOQ+1, LACK pointers respectively. TOQ is shiftable and loadable by LACK+1, BOQ+1 is only shiftable (advanced by a signal from *pktz-block*), LACK is only loadable (by new N(R) less 1).

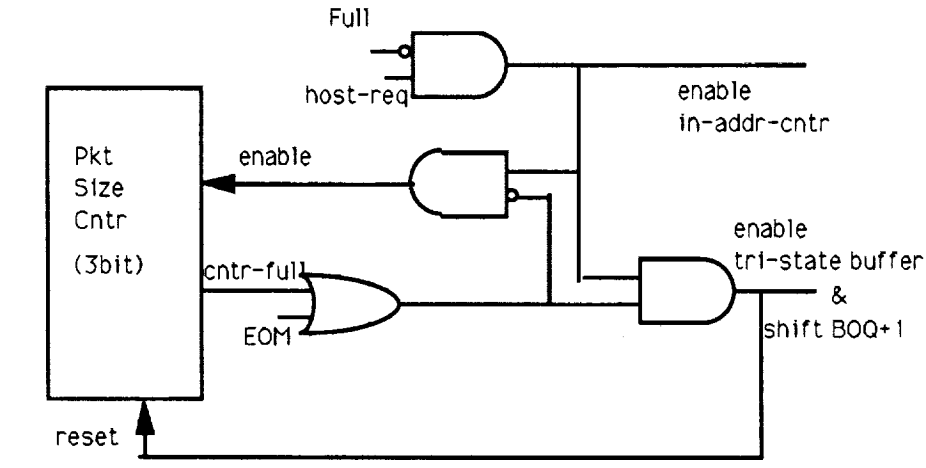
Comparators are used to generate the *Full* and *Empty* signals.

Go-Back-N Block: Figure 13a shows the structure of the Go-Back-N section. The N(S), N(R) are 4-bit shift registers, N(S) being loadable and the undecoded N(R) is glued beside the decoded N(R) (it is not part of the shift register).

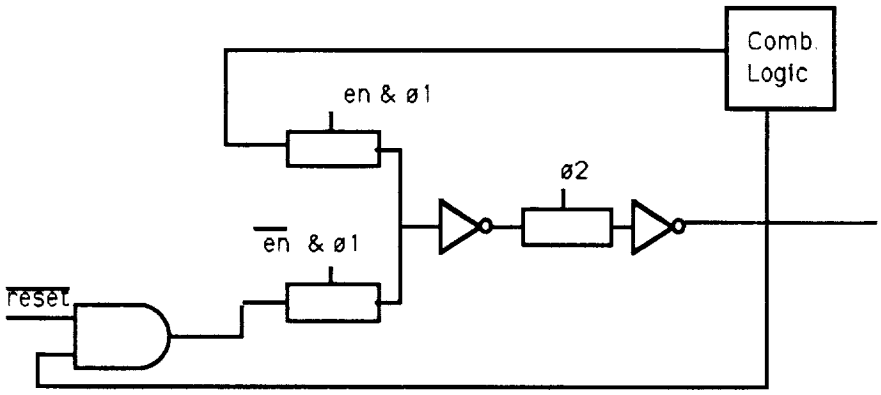
Timer Section: Figure 13b shows the internal structure of the timer section. The timer is a 4-bit counter with anenable signal. Figure 13c shows the schematics of the counter cell.

The rest of the circuit is really a small FSM to generate the *timeout* and *timeout(*2^{nd} *cycle)* from the *timeout(*1^{st} *cycle)* signal.

As mentioned before, there are a number of actions needed to be taken when timeout occurs and these cannot be done in one single cycle. Thus



(a)



(b)

Figure 11: Packetization Block

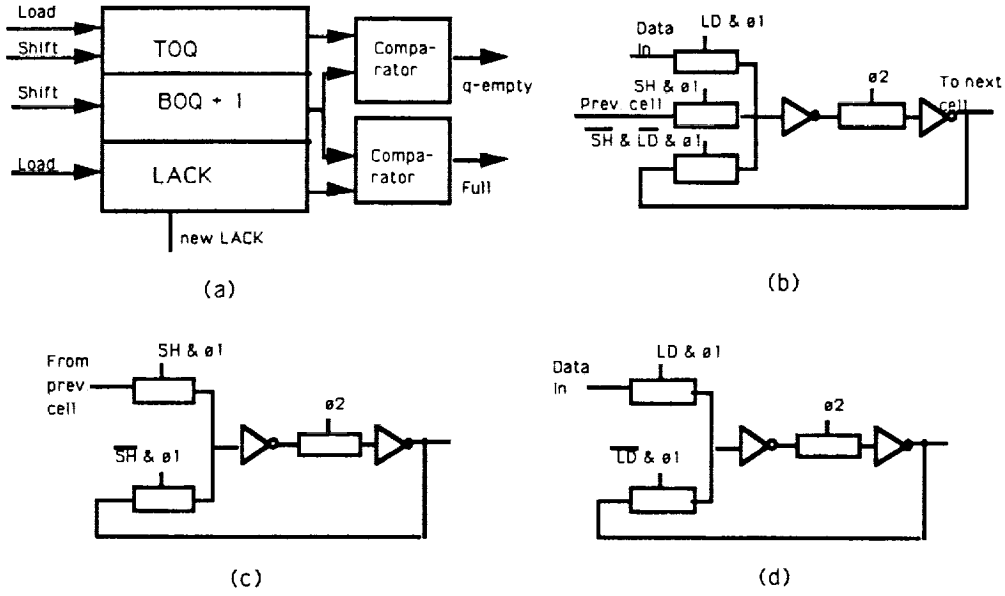


Figure 12: Pointers and Control Block

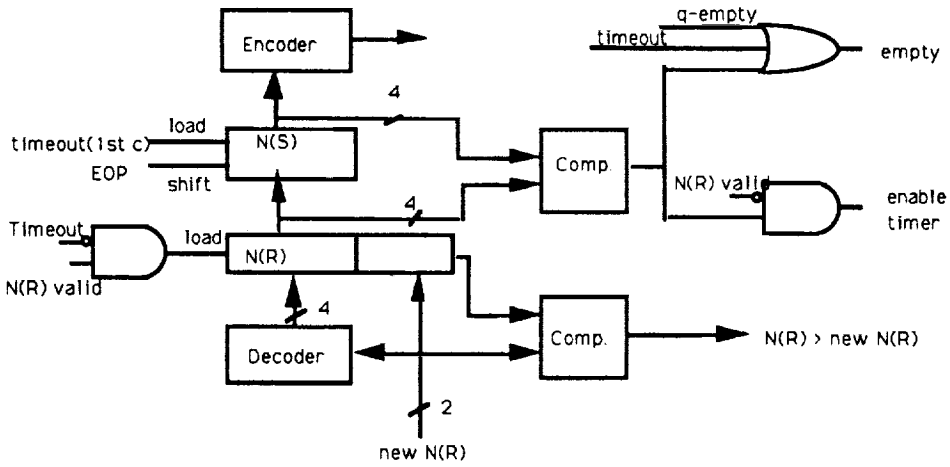
the small FSM is used to extend the timeout signal by one cycle. A timing diagram will illustrate the idea:

timeout(1^{st} cycle)	0000100...0100
timeout(2^{nd} cycle)	0000010...0010
timeout	0000110...0110

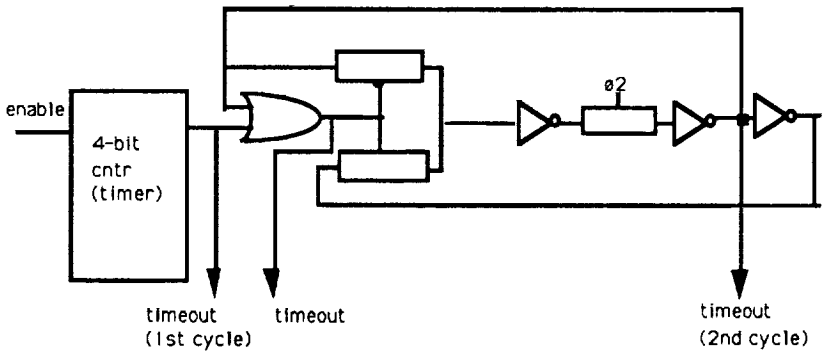
Notice that timeout(1^{st} cycle) is really the C_{out} of the most significant bit. The counter is either enabled or disabled but in both cases the C_{out} of the MSB will become zero in the next cycle. This is how we arrive at the above timing diagram.

3.2.2 Clocking

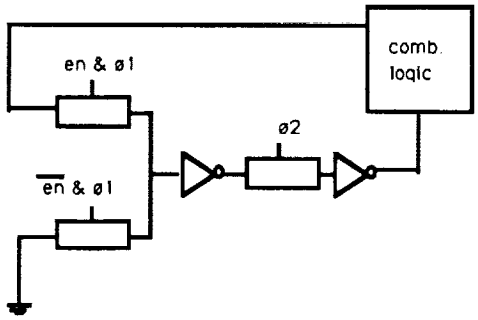
We adopt a strict clocking discipline in this project. All inputs and outputs of each block are stable Φ_1 except for the outputs of the address queue which are valid Φ_1. Furthermore, we have configured all our blocks in such a way that all combinational logic are performed during Φ_2. This facilitates easier



(a)



(b)



(c)

Figure 13: Go-Back-N block and the Timer section

testing and provides a natural scan path for LSSD. Figure 14 shows the clocking types of the inputs and outputs of various blocks.

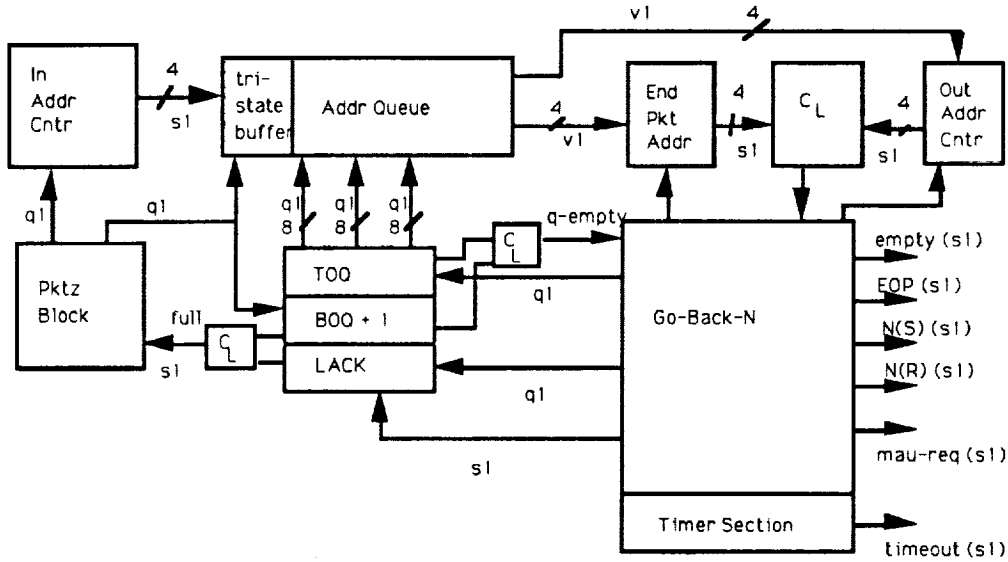


Figure 14: Clocking types of the various lines

3.2.3 Logic Descriptions of Datapath

The datapath of the system is given by the top part of Figure 14. The source of data is the *in-addr-cntr*. When there is a *host-req* and the queue is not full, the *in-addr-cntr* will start counting. When a packet has been received (i.e., current packet size is maximum or *EOM* is asserted by the host), the tri-state input buffer will be enabled and the end address of the packet will be stored into the FIFO queue according to the pointer BOQ+1, the first available position. Clearly, BOQ+1 should be updated.

On the output side, the *out-addr-cntr* will be enabled when there is a *man-req* and the queue is not empty (or no maximum number of outstanding packets). The value of this counter is constantly checked against the end address of the packet to generate the signal *EOP*. When *EOP* is true, the TOQ pointer is shifted and then the *end-pkt-addr* will be loaded. This operation requires two machine cycles so we have to assume that no packets

are shorter than 2 data words in order to maintain correct operation. There are also subtleties when the queue becomes empty. Figure 15b shows how the control signals to the output blocks are generated.

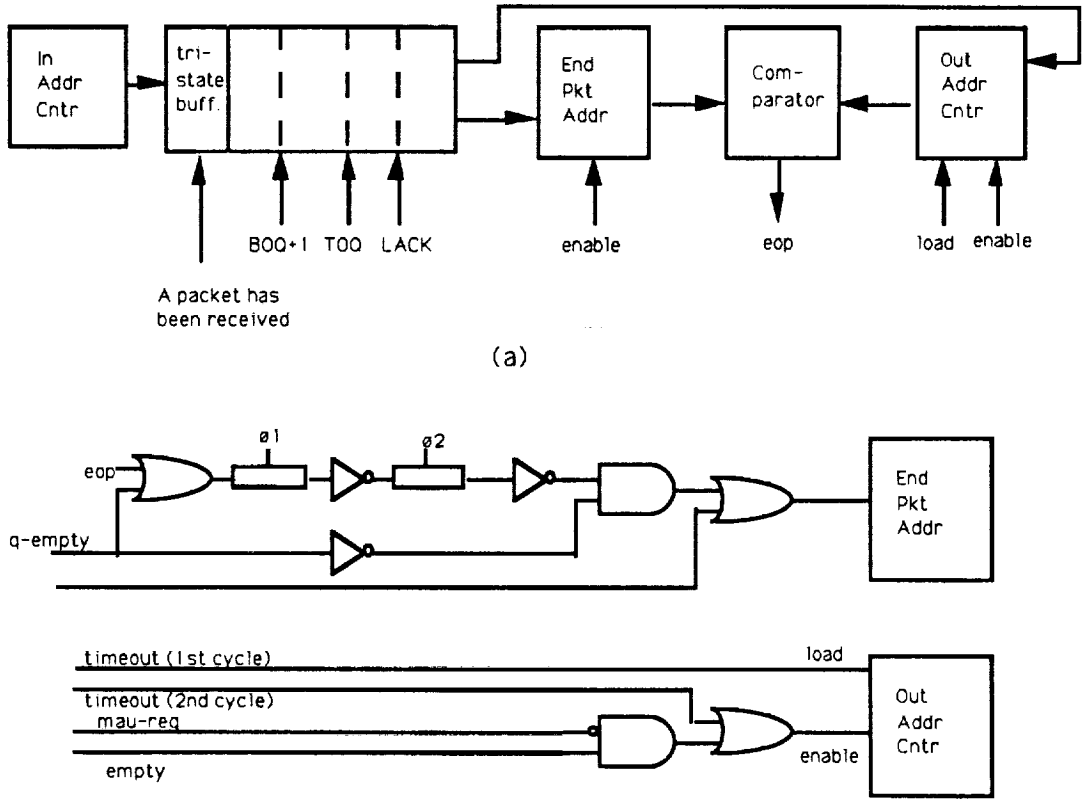


Figure 15: The datapath

Several remarks will be useful. The *in-addr-cntr* and *out-addr-cntr* always hold the next expected addresses. Thus, the host and media access unit can use these addresses without delay. The *Full* and *Empty* signals will show whether they are valid or not. Secondly, reading and writing in the same address queue location is not possible. Thirdly, the timeout actions can only be taken in 2 cycles. Thus, the three distinct signals timeout, timeout(1^{st} cycle), timeout(2^{nd} cycle) are all used to generate the necessary

control signals. A final point is that we need to load in the appropriate *end-pkt-addr* when the queue goes from empty to non-empty. The circuit diagram in Figure 15b shows how this is done.

3.2.4 Logic Description of Control Blocks

The control blocks shown in Figure 7 (lower portion) can be collectively considered as one large FSM. However, it can be broken down into a number of relatively simple functional units. We shall look at the functions, inputs and outputs of each unit.

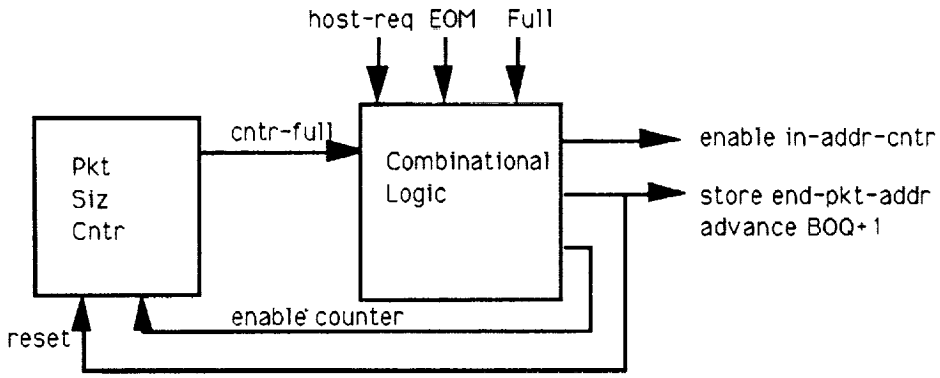


Figure 16: Packetization block

Packetization Block: This block is used to keep track of the size of the current incoming packet. When a maximum size packet is received or end of message arrives, the unit has to let the end packet address be stored into the address queue and advance the BOQ+1 pointer. Figure 16 shows the functions of this block.

The details of the combinational logic block is shown in Figure 11a. The *host-req* and *EOM* are external signals from the host. The *Full* signal is from the pointer control block.

Strictly speaking, there are 2^n possible internal states in this FSM where n is the size of the counter. However, as far as the inputs and outputs are concerned, there are only 2 states: counter full or not full. Thus, with the help of the counter, this machine becomes a trivial FSM.

Pointer and Control Block: This block is used to maintain the queue, i.e., to keep track of the top of the queue, bottom of the queue and the last acknowledged packet and to generate the *Full* and *q-empty* supervisory signals. Figure 17 shows the inputs and outputs of this block.

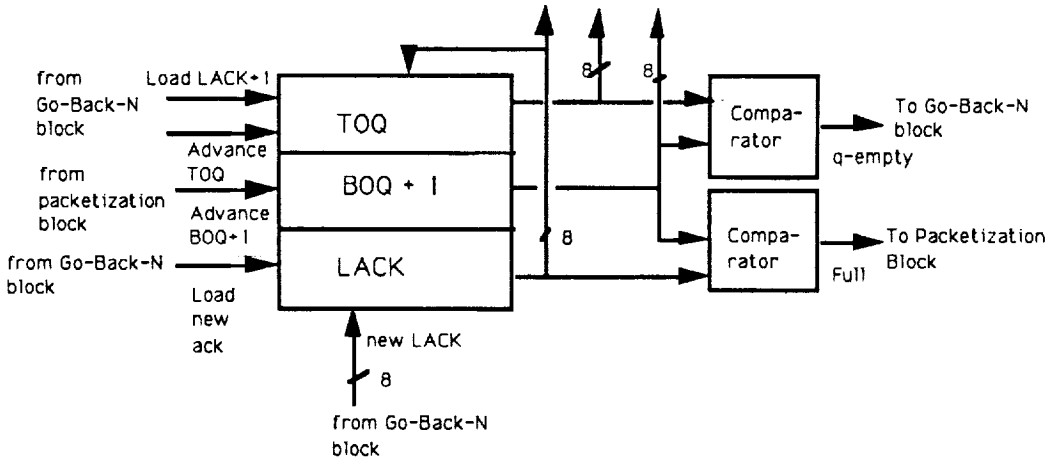


Figure 17: Inputs and outputs of the Pointer and Control Block

This is again a trivial FSM with only 3 states: queue full, queue empty or neither. When TOQ coincides with BOQ+1, the queue is empty. When BOQ+1 coincides with LACK, the queue is full.

Go-Back-N block: This is the most complicated among the 3 control blocks. It consists of 3 sub-blocks. Figure 18a shows these blocks. The first block is the output control block. Although it is more complicated, it is analogous to the packetization block. It generates the *Enable* and *Load* signals to the *out-addr-cntr* and *end-pkt-addr*. The second is the sequence number block. This block keeps track of the sequence numbers N(S) and N(R) and generates the appropriate control signals. This third block is the timer section which implements the timeout mechanism.

Figure 18b shows the inputs and outputs to the control section. It is just a combinational logic block where the logic is shown in Figure 13a.

Figure 18c shows the structure of the sequence number block. It is also a trivial FSM with 2 states: $N(S)$ equals $N(R) - 1$ or not equal.

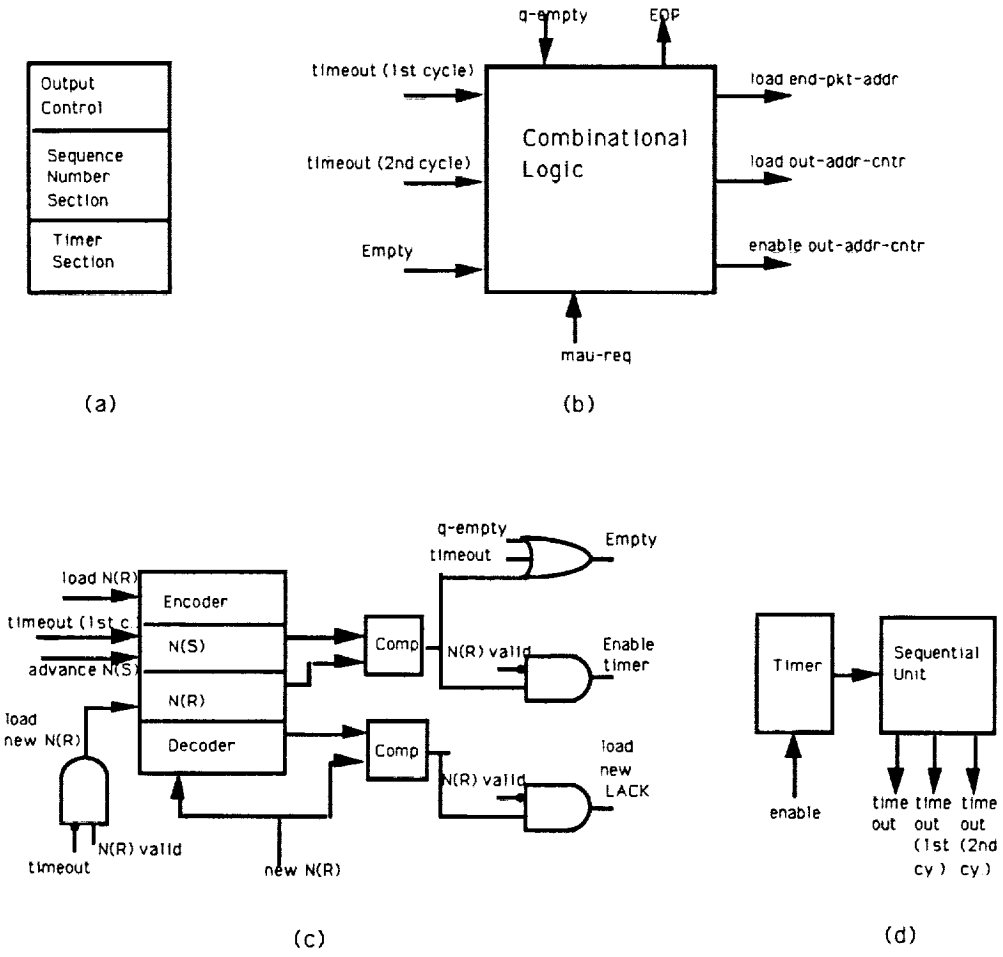


Figure 18: Go-Back-N Block

Figure 18d shows the input and outputs to the timer section. The sequential circuit is shown in Figure 13b. This timer has 3 states: no timeout, timeout(1^{st} cycle) and timeout(2^{nd} cycle).

4 Conclusions

We have presented our design of a high speed network interface unit (HSNIU) in this report. We have chosen to implement the packet processing functions in hardware, using VLSI technology, in order to satisfy our goal of achieving high rate on-the-fly processing for the multimedia communications applications while keeping the interface design relatively compact and at low cost.

The multimedia workstations that incorporate video and audio in addition to data and graphics will probably be very common in the near future. We note that the architecture and the operating system of these workstation also play an important role in achieving high bandwidth communication. As an example, in such a workstation, the ability to transfer an image from the video camera to the network adapter directly, without first transferring the data into the main memory of the workstation and then moving it to the network adapter would probably increase the performance significantly. Similarly, on the receiving side, it makes sense from the performance point of view to be able to move the received image directly to the frame buffer, without having to first copy it to the main memory. Thus, in the design of the multimedia workstations, such issues must be taken into consideration. With this goal in mind, we have been examining several existing computer architectures and have identified three basic types of architectures based on their bus structure. We are going to undertake timing analysis of multimedia workstation architectures, taking into consideration the effects introduced by the operating system. For example, it might make a significant difference whether the operating system allows a user process to directly access the network interface or not. We will then consider the integration of issues at both the system's architecture and the network adapter unit's architecture in order to determine the best configuration meeting the new objectives.

References

[1] H. Kanakia and F. Tobagi, "Performance Measurements of a Data Link Protocol," in *Proc. ICC'86*, (Toronto), 1986.

[2] G. Chesson, "The Protocol Engine Project," *UNIX Review*, pp. 70–77, Sept. 1987.

[3] H. Kanakia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," in *Proc. ACM SIGCOMM'88*, pp. 175–187, 1988.

[4] E. A. Arnould *et al.*, "The Design of Nectar: A network Backplane for Heterogeneous Multicomputers," in *Proc. ACM SIGCOMM'89*, pp. 205–216, 1989.

[5] D. Giarizzo, M. Kaiserwerth, T. Wicki, and R. C. Williamson, "High Speed Parallel Protocol Implementation," in *Protocols for High-Speed Networks* (H. Rudin and R. Williamson, eds.), pp. 165–180, IFIP, Elsevier Science Publishers B.V. (North-Holland), 1989.